OwnStream: Design and Implementation of a Scalable Peer-to-Peer Distributed Live Media Streaming

Servio Palacios, Victor Santos, Edgardo Barsallo, Bharat Bhargava

Department of Computer Science

Purdue University

West Lafayette, IN 47906

{spalacio, vsantosu, ebarsall, bbshail}@purdue.edu

Abstract

Peer-to-peer systems have been successful in implementing large scale media streaming systems. Unfortunately, there are no open-source solutions for peer-to-peer live video streaming (P2PLS). This paper focuses on such class of applications (e.g. an open source solution for distributed live media streaming). We propose Ownstream a peer-to-peer multimedia streaming based on a novel mix of algorithms and implementation of JavaScript's closures, promises, callbacks, and WebRTC for low-latency and high throughput. Ownstream implements its own object look-up via WebSockets and WebRTC. We choose WebRTC (e.g. WebRTC data channel) to solve the NAT-traversal problem and to obtain high-throughput. Ownstream restructures the topology dynamically in the case of network degradation. Ownstream offers authentication, privacy, and integrity of media chunks using public key infrastructure, a derivation of SSL protocol (e.g. DTLS) via WebRTC data channels, and a modified version of OTDP (One Time Digest Protocol) respectively. Ownstream implementation allows us to conduct experiments that demonstrates its high-quality video streaming in the presence of failures. We experimentally establish the applicability of Ownstream and provide the URL of Ownstream's GitHub repository.

Index Terms

Cloud computing, event-driven model, media streaming, JavaScript, Peer-to-Peer, WebRTC

I. Introduction

Commercial scale video systems have demonstrated the market's momentum via a mixture of content attracting an enormous amount of users. Peer-to-peer (P2P) systems constitute the backbone in a diversity of distributed implementations (e.g. video streaming, file sharing) due to their resiliency and scalability.

We propose Ownstream a new distributed peer-to-peer video streaming system. Ownstream takes advantage of emerging technologies in order to accomplish a desired operability. Ownstream offers media streaming capabilities on top of WebSockets, WebRTC, and JavaScript. Challenges have been identified in [1], [2]. For instance, how to deal with peer selection in a high churn and heterogeneous network. Ownstream offers authentication and integrity capabilities. For instance, We authenticate every peer against a centralized server, *supervisor*, using an authentication protocol based on TLS and DTLS cookies [3]. Once authenticated, peers are monitored including their connection channels using WebSockets heartbeats and WebRTC data channel features. In the case of network degradation, the system is capable of restructuring the topology dynamically, changing active *receivers* or *bridges* to obtain the performance in the overall network similar to [1], [2]. We provide future improvements and capabilities expansion.

A brief explanation of the building blocks of the system is in Section II. The key concepts and architecture of the system are in Section III. We provide an extensive set of experiments demonstrating the feasibility of our system in Section IV. Section V provides an insight into challenges experienced throughout the development of the system and future work. We include a discussion of related research and existing solutions (Section VI). Finally, we present our conclusions in Section VII.

II. Background

A. WebSockets

WebSocket enables bidirectional, message-oriented streaming of text and binary data between client and server. The API provides the following services [4], [5], [6]:

- Connection negotiation and same-origin policy enforcement.
- Interoperability with existing HTTP infrastructure.
- Message-oriented communication and efficient message framing.
- Subprotocol negotiation and extensibility.

We use SocketCluster framework [7] to provide WebSocket services. SocketCluster supports both direct client-server communication (similar to *socket.io*) and group communication via pub/sub channels. In addition, this library implements features such as heartbeats, timeouts, support for automatic reconnects, and multi-transport fallback functionality as recommended in [5], [6]. SocketCluster is designed to scale both vertically across multiple CPU cores and horizontally across multiple machines/instances.

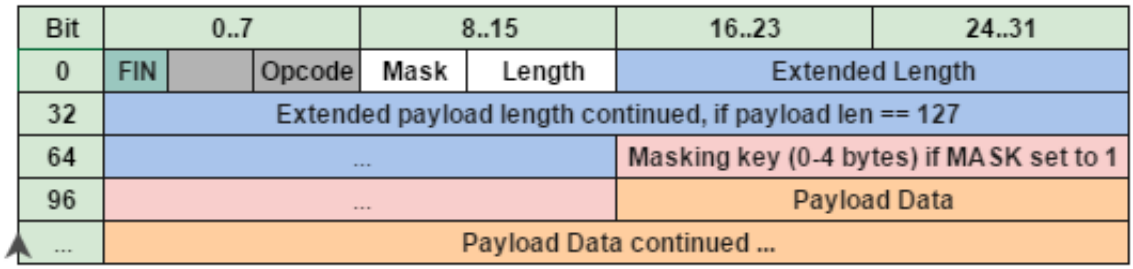

Fig. 1: WebSocket frame [5]

1) WebSocket frame: Figure 1 shows a map of a typical WebSocket frame for the RFC6455 [8]. In detail, the first bit FIN indicates whether the frame is the final fragment of a message. The opcode (4 bits) indicates the type of transferred frame: text (1) or binary (2), (10) for connection liveness checks.

2) Deploying WebSocket Infrastructure: Since we do not have control over the policy of the $client's$ network, we have to use TLS by tunneling over a secure end-to-end connection. Therefore, WebSocket traffic can bypass all the intermediate proxies and firewalls. We can send security parameters to SocketCluster to allow secure connections.

3) Performance objectives [5], [6]:

- Use secure WebSocket (WSS over TLS) for reliable deployments.
- Optimize payloads to minimize transfer size.
- Consider compressing UTF-8 content to minimize transfer size.
- Monitor the amount of buffered data on the client.
- Split large application messages to avoid head-of-line blocking.

4) Performance Notes [5], [6]: We take into consideration the performance criteria based on WebSocket architecture. For instance, all WebSocket based messages are delivered in the same order in which they are queued by the client. Queued messages or large messages will delay

delivery of messages queued behind it (e.g. *head-of-line blocking*). The application layer can split large messages into smaller chunks or implement its own priority queue. As a result, since the application is delivering latency-sensitive data, we take care of the payload size of each message.

B. WebRTC

Web Real-Time Communication (WebRTC) is a collection of standards, protocols, and JavaScript APIs, the combination of which enables peer-to-peer audio, video, and data sharing among browsers (peers) [6]. WebRTC transports its data over UDP because latency and timeliness are critical [6].

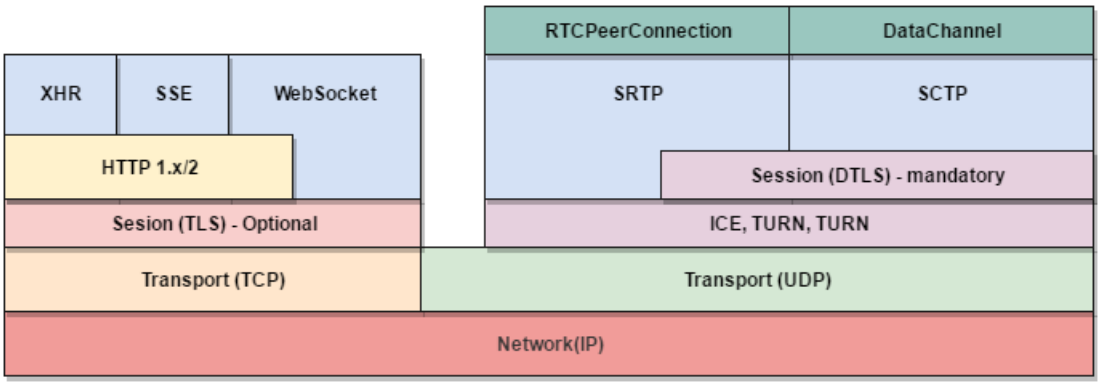

Fig. 2: WebRTC Protocol Stack [5]

1) WebRTC protocol stack components: Figure 2 shows the protocol stack.

- ICE: Interactive Connectivity Establishment.
- STUN: Session Traversal Utilities for NAT [9].
- TURN: Traversal Using Relays around NAT [10].
- SDP: Session Description Protocol [11].
- DTLS: Datagram Transport Layer Security [12]
- SCTP: Stream Control Transport Protocol [13]
- SRTP: Secure Real-Time Transport Protocol [14]

ICE, STUN, and TURN are necessary to establish and maintain a peer-to-peer connection over UDP. WebRTC enforces secure data transfers between peers; therefore, DTLS is used to encrypt data transfers [5], [6]. On the other hand, SCTP and SRTP are used to multiplex the different streams, provide congestion control, and give partially reliable delivery on top of UDP.

2) RTCPeerConnection API: RTCPeerConnection encapsulates all the connection setup, management, and state within a single interface.

3) DataChannel: DataChannel API enables the exchange of arbitrary application data between peers. Each data channel can be configured to provide the following:

- Reliable or partially reliable delivery of sent messages.
- In-order or out-of-order delivery of sent messages.

The channel can be configured to be "partially reliable" setting the maximum number of retransmissions or setting a time limit for retransmissions. The WebRTC stack will handle the acknowledgments and timeouts.

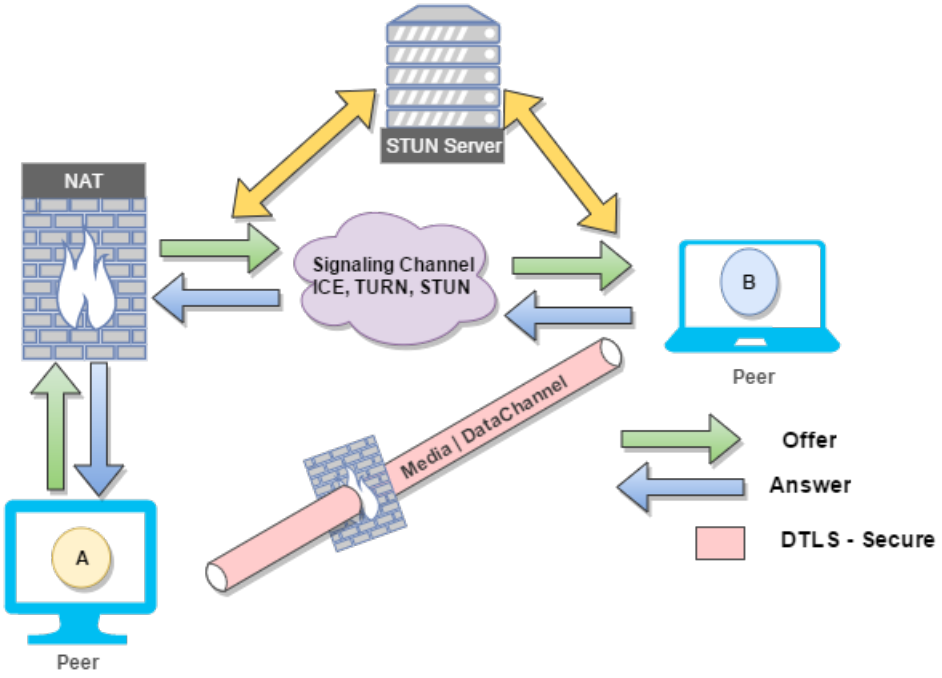

Fig. 3: NAT traversal [5]

We deal with NAT traversal problem using WebRTC capabilities, as shown in Figure 3. In addition, we wrapped PeerJS [15] library to provide WebRTC basic functionality and functions (e.g. Communication Layer). The built-in ICE protocol performs the necessary routing and connectivity checks. The delivery of notifications (signaling) and initial session management is left to the application. This signaling is handled using PeerJS [15] library integrated with our communication and security layer.

4) Session Description Protocol: WebRTC uses Session Description Protocol (SDP) to describe the parameters of the peer-to-peer connection [1]. SDP does not deliver media itself; instead, it is used to describe the "session profile". SDP is a simple text-based protocol for describing the properties of the intended session. WebRTC applications do not have to deal with SDP directly because the JavaScript Session Establishment Protocol (JSEP) abstracts all the inner workings.

5) Interactive Connection Establishment (ICE): The peers must be able to route packets to each other in order to establish a peer-to-peer connection. Peers can be in different private networks which will cause the peer-to-peer connection to fail. WebRTC manages this complexity through ICE agents. As shown in Figure 2, each RTCPeerConnection connection object contains an "ICE agent".

- ICE agent is responsible for gathering local IP, port tuples (candidates).
- ICE agent is responsible for performing connectivity checks between peers.
- ICE agent is responsible for sending connection keep-alives.

III. Design and Implementation

A. Architecture

Ownstream is composed of daemons installed with just one command via the Node Package Manager [16]. The system has a peer-to-peer architecture which comprises three main components, as shown in Figure 4. Each one of the components is described as follows.

- *Broadcaster*: The broadcaster is the source of a particular media stream. This is a multiplatform [17] application which leverages cutting edge web technologies such as WebRTC, Media Source Extensions [18], and all experimental components included in the Chromium [19] browser. With all Node.js native capabilities, the *Broadcaster* extract binary data either from a camera/microphone hardware or a pre-encoded WebM [20] container, which can be audio or video. Finally, the broadcaster streams the source media via WebRTC data channels to the neighbors (e.g. immediate connected peers).
- *Receiver*: The receiver is an *HTML5* web application which receives the desired media stream. Using several technologies present in the *Broadcaster* (WebRTC and Media Source Extensions), the *Receiver* consumes a stream and **relays** the stream to its neighbors peers.
- *Supervisor*: The supervisor registers each peer (Broadcaster or receiver). The supervisor monitors every connection to the components in real time and notifies every relevant

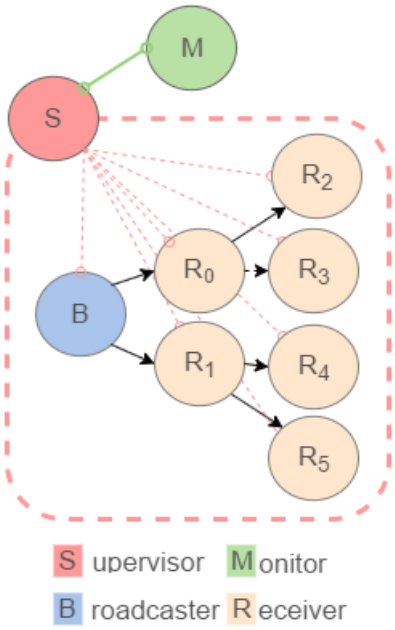

Fig. 4: OwnStream Architecture Diagram

change to the corresponding nodes, i.e. *disconnection of a peer*. We use the supervisor as Certification Authority (CA), to take care of the authentication of every peer facing challenges similar as [3].

After a *Broadcaster* creates a stream, the *supervisor* registers and announces the existence of this new stream to the *Receivers*. When a receiver chooses to join the stream, the *supervisor* registers the new component in the stream topology. When the new receiver is inserted in the virtual topology, signaling events and connection setup is made to directly connect this new peer to its neighbors. It creates a path where the stream will flow and deliver the content. Currently, we have implemented two topologies: **Linked List**, where viewers are chained one after another in the order of arrival. **Binary Tree**, where viewers connect into a tree structure. The latter has the clear advantage of the scalability and low content delivery latency. The stream only takes the *depth* of the tree to reach the last node in the topology, and the scalability of the overall topology follows the geometric series:

$$2^{n+1} - 1$$

where n is the depth of the tree.

An additional component, not part of the architecture, is used for administrative and experimental purposes.

- *Monitor*: The monitor is an *HTML5* web tooling interface used to visually emit commands to *Receivers* and evaluate the effectiveness of a particular streaming topology. Some of the features of the *Monitor* is to measure stalled stream events, disconnections, apply different failure distributions (see section IV), and visualize the shape of the topology (normally a binary tree).

The system relies on WebSockets and WebRTC data channels. We use the SocketCluster [7] library for the real-time signaling communication from which we leverage convenient features such as heartbeats, connection state, and event-driven communication. The *supervisor* is continuously listening for connections from *broadcasters*, and *receivers* (e.g. peers). At each disconnection, the supervisor updates the data structure to balance the load among peers. In addition, the supervisor serves as a low latency fault tolerance messenger and certification authority.

B. The Communication Layer

We now explain the communication components that provide signaling features during a media streaming session. The communication layer abstracts all the burden of WebRTC, SocketCluster and the Security Library. This layer provides application transparency to the operations being executed under the hood.

1) *Collision Resistant Unique Identifier*: Since we wrapped PeerJS library inside of the communication library, we need to provide a collision resistant function to generate unique names with negligible probability of collisions. We use a randomly generated UUID which has 128 bits. The chance that 128 bits of having the same value can be calculated using probability theory (birthday problem), which can be shown to be negligible.

$$p(n) \approx 1 - e^{\frac{-n^2}{2*2^x}}$$

   We use this *PeerID* to ensure unique WebRTC data channels between peers.
2) *Public and private key*: Every peer has its own RSA private and public key. We provide a parameter to configure the level of security enforcement among peers (e.g. key bit length ≥ 2048). These keys are used for the Authentication Protocol described in the Security Layer. The private key is never exposed outside of the system.
3) *Connected Peers*: Every peer keeps its neighbor set similar to [21], [22]. We then forward packets received or send to the application layer, the *video chunks* received.

4) *Signaling Capabilities*: Every peer connects to the *supervisor* using the communication library. Consequently, this layer handles all the signaling, heartbeats, and disconnections.
5) *WebRTC Data-channels*: We want to be able to create a variety of topologies using our implementation. Hence, the communication library provides a capability that is the equivalent to creating an *edge* in a graph. Consequently, we can create a rich set of topologies using the basic building blocks of *edges* (e.g. data channels) and *vertices* (e.g. peers, receivers or broadcasters).

C. Security Layer

In this subsection, we discuss the security guarantees that our system offers.

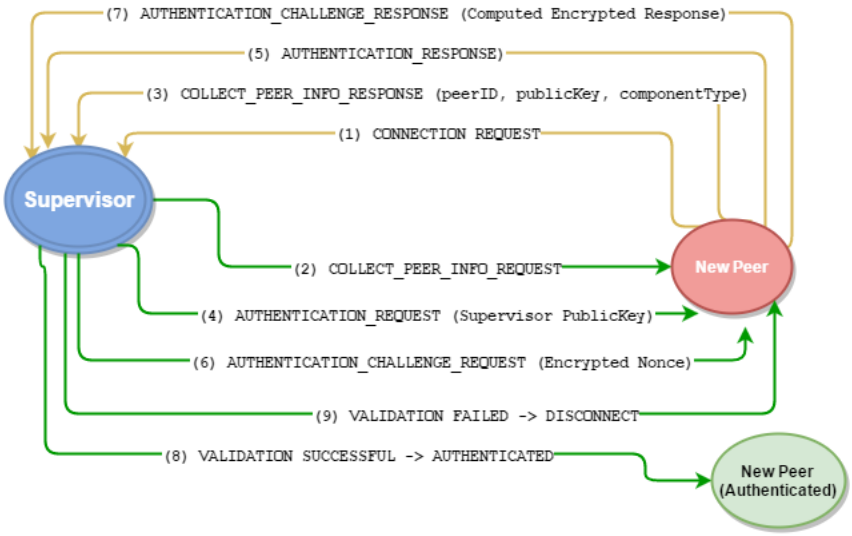

Fig. 5: Authentication Finite State Machine

1) *Authentication Protocol*: We have implemented a handshake protocol, as shown in Figure 5. We took some ideas form TLS and DTLS [3].
   - We receive a socket connection to our SocketCluster server.
   - The supervisor sends the message `COLLECT_ PEER_INFO_REQUEST`.
   - The peer replies with a `COLLECT_PEER_ INFO_RESPONSE`, including its PeerID (used in the PeerJS connection), the Public Key, and a componentType (Broadcaster or Receiver).

- The supervisor sends its public key to the peer using `AUTHENTICATION_REQUEST` message. Our future work will include a Certification Authority to deal with digital signatures and public keys.
- The peer ACKs with `AUTHENTICATION_RESPONSE` message.
- The supervisor now computes a nonce including two numbers and encrypts those numbers using the peers public key. We the send the nonce using `AUTHENTICATION_CHALLENGE_REQUEST`.
- The Peer receives the challenge request, decrypts the nonce, and computes a number based on two random numbers included in the nonce. The peer then proceeds to encrypt the response using the supervisor's public key. Finally, the peer answer with `AUTHENTICATION_CHALLENGE_RESPONSE`.
- The supervisor now checks the response given by the peers. If matches the one computed on the server, then the peer has now the state of `AUTHENTICATED` and is ready to initiate or receive streams.

2) *Integrity Validation of Video Chunks*: In order to provide data integrity during peer-to-peer video streaming, we use a similar approach to the one described in [23]. Ownstream's implementation has some major differences. First, we only have a *broadcaster* for a specific stream. Second, we have implemented our own object (stream) look-up. Third, our implementation is live-oriented and we do not know all video-chunks in advance. We do not assume that connecting peers are trustworthy, so we enforce authentication through our *supervisor* including a Public Key Infrastructure (PKI). Every *authenticated* peer is allowed to be part of our streaming sessions. We compute message digests using a keyed SHA-256 (*HMAC*) and NodeJS Crypto library instead of digital signatures as in [23], but we use video chunk VC instead of segments.

$$D_{ij} = h(K_i, VC_j)$$

where D_{ij} is the computed HMAC for the video chunk (VC) *j* and K_i is the *supervisor's* generated key for stream *i*.

We modify the steps of One Time Digest Protocol (OTDP) in order to adapt it to our design and implementation as follows:

- The peer P_0 authenticates itself against the *supervisor*.

- P_0 requests a list of media streams to the *supervisor*.
- The *supervisor* provides a set of keys based on the search results.
- When the *broadcaster* (P_i) starts the streaming, it sends an initialization packet to P_0.
- The *broadcaster* (P_i) sends both data and digests to P_0.
- P_0 verifies random video chunks. Furthermore, P_0 can verify every packet depending on the *threshold* defined as a parameter.

The *broadcaster* computes the hash of every video chunk and sends the packet via the data channels connected to its *receivers*. Next, the *receiver* checks for the integrity of the packet and compares with the digest sent by the *broadcaster* or source of the media stream. In contrast to [23], we do not know all chunks of media in advance. Therefore, we cannot compute random message digests and send those to every peer in our topology. We compute message digests of the chunks generated by the application layer. The *receiver* can verify at random one message digest to identify forgeries. We implemented a probabilistic approach using a uniform random number generator and comparing those generated numbers against a *threshold* defined as a parameter. The size of the message digests are 44 bytes (Base64 encoded).

D. Limitations

The system has several limitations that are mainly due to client-server nature of web technologies. Currently, we can use two execution points: Any browser's ECMAScript run-time with WebRTC support, where users go to a website and automatically setup all the execution environment, and a Node.js application (supervisor) that currently runs on top of the V8 engine with I/O and OS's system call capabilities. The browser run-time is limited to its intrinsically high failure rate, especially in mobile devices. Consequently, the lack of fault tolerance features is its main limitation, specifically in the *supervisor*. Although we offer authentication and data integrity protocols, security poses a significant challenge because of nature of the inherently heterogeneous and high-churn network.

IV. EXPERIMENTS

Our experiments were aimed to evaluate two main aspects of the system. First, we injected failures to simulate how users would abandon the stream breaking the topology causing stalled playbacks, and lost packets. Second, we evaluated the performance of our security scheme.

Media Information		
General	Format	WebM
	Format version	Version 2
	File size	24.0 MiB
	Duration	2mn 53s
	Overall bit rate mode	Variable
	Overall bit rate	1159 Kbps
Video	Format	VP8
	Width	1920 pixels
	Height	1080 pixels
	Display aspect ratio	16:9
	Frame rate mode	Constant
	Frame rate	29.970
Audio	Format	Vorbis
	Bit rate	128 Kbps
	Channel(s)	2 channels
	Sampling rate	44.1 KHz

TABLE I: Media Information

A. Experiment Setup

The experiment setup consisted of 80 machines with different specifications, usually 4 to 8 2.0 GHz cores, 8GB of RAM, all connected in a 1Gbps local area network. All 80 machines are used as receivers. In another machine, we have the *supervisor* and the *monitor*. Finally, we stream the media from a separate machine running the *broadcaster*, this sums up a total of 82 machines for all experiments. In table I, we can see the description of the WebM video file used for all experiments.

The *monitor* component has the capability of visualizing and manipulate the topology of a stream. We are specially interested in setting up a stream using the 80 machines (81 including the *broadcaster*), and induced failures in order to observe the behavior and quality of service of our system.

B. Method of injecting failures

We define a failure as a disconnection of a viewer in the topology, which will break the flow of the stream. We then, apply different rates of failures in the a tree topology to simulate

viewers leaving and joining the stream. The rates are 1 to 5 failures per second. To better evaluate the behavior of the system in different failures scenarios, we choose from five probability distributions to simulate each scenario:

- Binomial.
- Uniform.
- Beta.
- Chi-Squared.
- Poisson.

At the moment of inducing a failure, we generate a random number based on one of the previous distributions. For each distribution, we ran 5 experiments, each one consisting of 1 minute of the stream across the 80 machines, each experiment with 1, 2, 3, 4, and 5 failures per second. For each distribution and failure rate, we took 4 samples in order to rule out spontaneous network congestion and processing load fluctuations. It is important to mention that each time a viewer fails (killed), it is disconnected and reconnected to the topology as a new incoming viewer.

The metrics used to evaluate the behavior of the system were:

- *Stalled:* Overall stalled playbacks in the stream during the experiment.
- *Total Stalled Time:* Overall playback stalled time during the experiment.
- *Average Stalled Time:* Overall average stalled time during the experiment.

These metrics all together will describe the quality of the stream on different failure rates and scenarios.

C. Results

Figures 6-10 show the results of the stalled counter and total stalled time for each distribution, varying from 1 to 5 failures per second. As we can see in figure 6, the **uniform** distribution shows a gradual increment in stalled counter and total stalled time as failure rate increases. Surprisingly, the stalled counter and total stalled time start to decrease after the 4 failures per second. One explanation for this behavior is that machines are getting disconnected at a higher rate than they return to the topology, therefore reducing the number of connected machines at any time during the 1-minute experiment. The lowest stalled counter and total stalled time for the uniformly distributed failures experiment are 36 and 28 seconds respectively for 1 failure per

second. The highest are for 4 failures per second, with 152 stalled playbacks and 127 seconds of total stalled time.

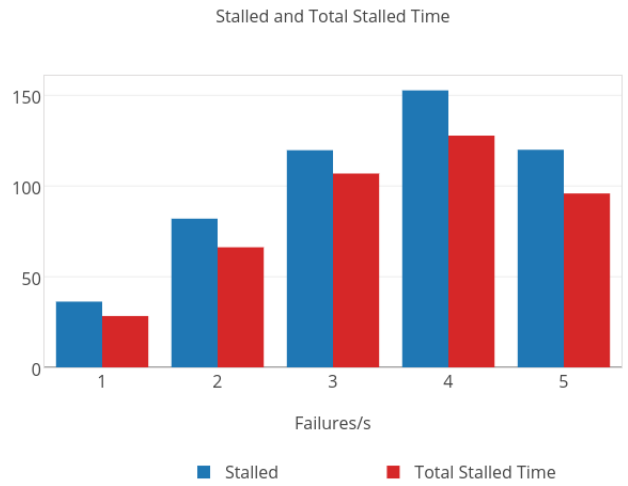

Fig. 6: Uniform Distribution, Stalled and Total Stalled Time

Figure 7 shows the results for the **binomial** distribution. This particular distribution induces failures at the center of the Tree topology. We can observe that both stalled counter and total stalled time are very low compared to those in the uniform distribution experiments. At some rates (2 and 4) there were no stalled playbacks at all. The binomial distribution induce failures mostly at the center of the topology tree. The low stalled counter may be explained by analyzing the following scenario: A broadcaster send a chunk of the media through all the topology at time T_1, and let say that this media chunk reaches the leaf nodes in the topology after n seconds of latency. If the binomial failure experiment induces failures at the center of the topology, it may be common that the chunks traveling the topology are in transit before the center, or after the center at the moment of failure. Since the chunk, if already delivered to the leaves or are before the center, there won't be stalled playbacks. Each random number was generated with 20 trials and 0.80 of success probability.

In figure 8 we can observe how the system behaves with failures induced using a **Poisson** distribution. The figure is similar to the Uniform experiment, but we can see much lower *stalled counter* and *total stalled time*. In the sense of the location of the induced failure in the topology, this distribution behaves similar to the binomial but closer to the root instead of the middle of the tree. Each random number was generated with a mean of 80/2 (half number of nodes in the

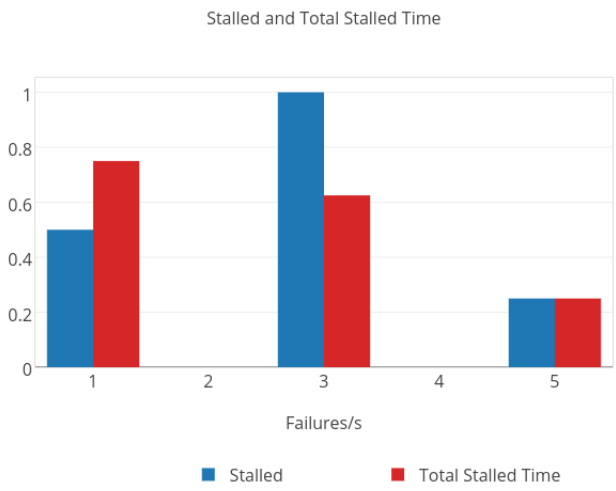

Fig. 7: Binomial Distribution, Stalled and Total Stalled Time

topology).

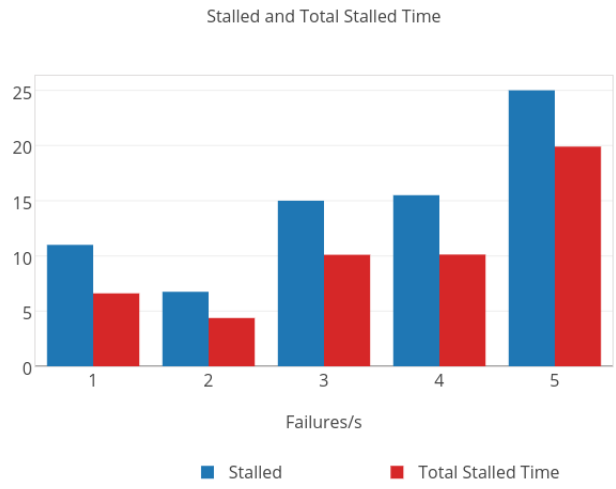

Fig. 8: Poisson Distribution, Stalled and Total Stalled Time

Figure 9 shows how the **beta-induced** failures affect the stream quality. This distribution will induce failures right next to the root (*broadcaster*). The results indicate the largest amount of stalled playbacks and total stalled time of all distributions. One possible explanation for this is that since the failures occur next to the broadcaster, many chunks of the media are lost at the very first nodes of the topology, propagating the delay to the entire topology and causing mass

stalled playbacks. Each random number was generated with $alpha = 0.5$ and $beta = 3$.

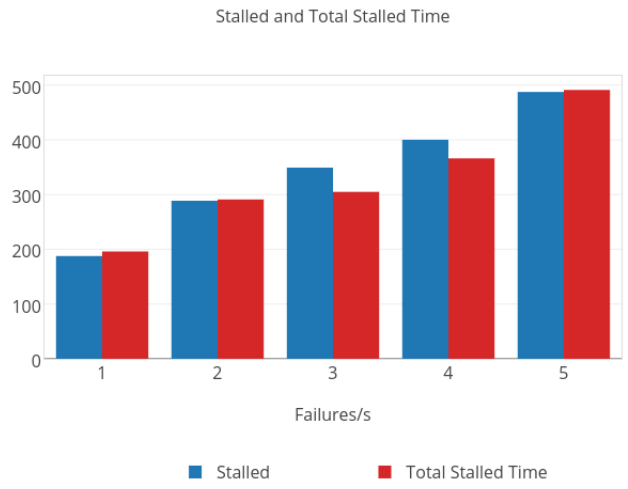

Fig. 9: Beta Distribution, Stalled and Total Stalled Time

Finally, the results for failures induced by a **chi-square** distribution are shown in figure 10. As we can observe, the results are very similar to the binomial distribution in terms of both *stalled* counter and *total stalled time*. Each random number was generated with 5 degrees of freedom.

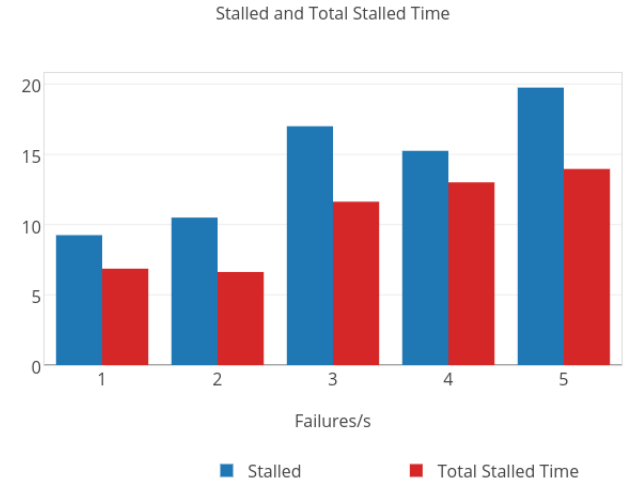

Fig. 10: Chi-Squared Distribution, Stalled and Total Stalled Time

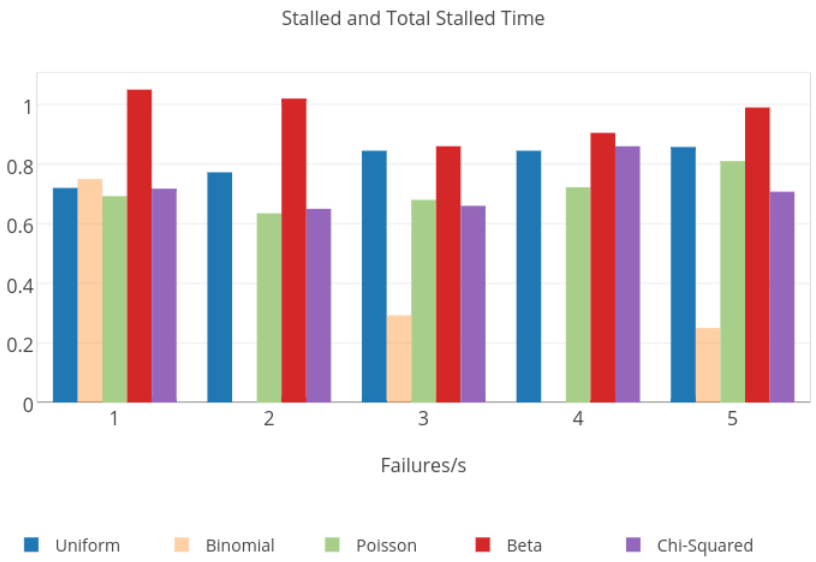

Fig. 11: Average Stalled Time per Distribution

D. Analysis

The main contribution of our work is to expose a proof of concept of a decentralized (in the sense of content distribution) of a media content streaming using peer-to-peer communication. Also, we demonstrated that it is totally feasible to maintain a considerably good quality in exchange for resource cost and network overhead. All five experiments with different distributions and failure rates aimed to simulate a real life failure scenarios, and how the system will behave in such conditions. So far, we have shown in figures 6-10, how the streaming quality is affected and how stalled playbacks scale as failure rate increases. In figure 11 we show an interesting aspect which is consistent no matter what distribution is applied to simulate failures. The average stalled time in the overall topology is below 1 seconds in most scenarios. This means that the overall stream interruptions in the topology are not longer than one second, and in most cases are milliseconds which can be tolerated and perhaps be so brief that are not even perceived by the viewer. The only case when the average stalled time goes drastically high is when the failure distribution is similar to a beta distribution. In this case, failures occur close to the root and causes severely stalled playbacks, which propagates through all the topology.

E. Potential alternate use cases

We have shown how we maintain considerable good quality in a live stream topology using mainly 3 components: *supervisor*, *broadcaster*, *receivers*. We are presenting our system in the context of live media streaming. On the other hand, there is no reason why this same concept cannot apply to other scenarios which also relies on CDN to distribute content. Some of the potential scenario to apply the same technique are:

- *Mobile app content distribution*: Recently, many mobile apps(especially social media), distribute media content in a graph fashion. I.e. friends to friends. This case will be even simpler than our scenario since the stream of content is not real time. Implementing a similar architecture would greatly reduce the cost of scale-out large mobile apps which needs to distribute media content.
- *Software Updates*: Modern software, such as those built with Electron [17], supports hot updates, meaning that the application can be downloaded in the background and installed inside a sandboxed installation without the need of natively uninstall and install a different binary. The same concept can be applied to distributed hot updates without requiring large content distribution networks.
- *Distributed Web Pages distribution*: In a booming world wide web, denial of service is a very common problem. When a site is experiencing a heavy load of requests, the server(s) starts to deny service. This may be avoided by implementing a similar architecture as our system. If many clients are on the website, connected via a persistent WebSocket, the content of the web page can be cached in the current clients, and when additional requests arrive, a peer-to-peer channel can be created in order to let the existing clients to serve the web page content instead of putting all the load on the web servers.

F. Security Layer Overhead

Another vital part of the system is its security layer. In the experiments, we demonstrate that RSA key pair generation behaves as shown in Figure 12. In the implementation, we have used 1024-bits keys in order to provide authentication. However, it is possible to use 2048-bits keys or bigger to provide stronger security. In addition, figure 13 shows the computation overhead of the most important methods of our security layer . It is notable that keyed hash function generation time is similar to the SHA-256 digest generation. Nonces generation is negligible compared with other measures (e.g. just 20 microseconds or less). Finally, integrity check adds

9.436 milliseconds on average (each side *broadcaster* and *receiver*) to the overall video chunk generation and transmission.

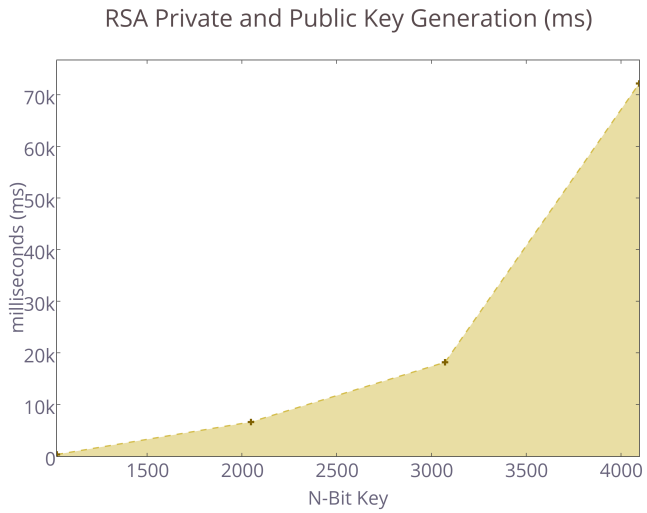

Fig. 12: RSA Key Pair Generation

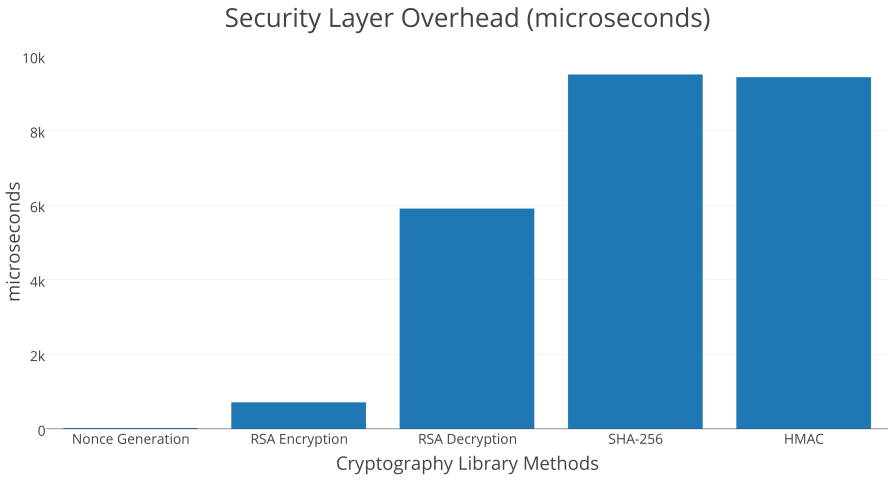

Fig. 13: Security Layer Overhead

V. Future Work

In the experiments section, we showed the potential of our system. However, the system should have many other qualities to be considered robust. In this way, we have a tight schedule towards the next version of **OwnStream**. To further improve the quality of the streaming and prevent stalled playbacks, the following ideas may be considered.

- *Fault Tolerance*: As we previously discussed, we want to provide fault tolerance guarantees by replicating *supervisors*. SocketCluster balances the load among many workers, but we will need a layer to provide supervisor's high availability even in the case of failures.
- *Forward Relay Points*: Certain vertices in the topology can be used as 'stable' relay points. Some metrics such as stable time, bandwidth, and reliability overall can be used to decide which vertices to connect further in the topology and bypass several hops. This may help to prevent stalled streams and delay in the stream if a path is congested or many failures have occurred.
- *Redundant connections*: Redundant connections to successors and predecessors in the topology can be established as backup connections in case of failure. This may hide the current topology reconnection procedure since the connections are already present between the next or previous vertices.
- *Self-balancing topology*: Using stability and performance metrics, we can use a self-balancing topology which will move vertices back and forth in the topology in order to maintain a smooth stream, i.e. prevent bottleneck links and colocate nodes according to geographical position. This, of course, is a very complex approach and have undesirable consequence since moving vertices in a live streaming flow can affect the flow itself.
- *Triple Branched Broadcaster*: It may be a good idea to create 3 different branches in from the broadcaster instead just one big topology. For example, we can create 3 different branches, and each branch streams a low, medium, and high-quality stream respectively. This approach may alleviate bottlenecks and too heavy heterogeneity in the capacities of the vertices and edges in the topology.
- *Security*: Security and privacy are a concern. We need an exhaustive analysis of the different scenarios where our system could become harmful or a threat to the user's privacy or system. For instance, we will provide a solution for the DTLS handshake self-signed certificates in WebRTC clients using the *supervisor* node to provide another level of security within our system. In addition, we need to overcome WebRTC functionality in restricted networks, to accomplish this we need to implement a tunneled service that serves as gateway supporting ICE and STUN protocols as in [24]. We have implemented an Elliptic Curve Cryptography module, future versions will let user choose between different security methods.
- *Benchmarks*: Finally, we want to evaluate how well **OwnStream** can challenge existing commercial media streaming systems.

VI. Related Work

Ownstream shares some characteristics as Peer2View [25]. For example, Peer2View is built on top of UDP-based transport library which provides reliability and security guarantees. In addition, they also authenticate their users against a central authority (e.g. in our case the supervisor). They provide encryption using SSL and integrity. However, there are many differences between OwnStream and Peer2View, the latter solves the NAT problem with NATCracker [25], the former uses WebRTC, ICE, STUN and TURN. Moreover, Peer2View is a commercial peer-to-peer live video streaming (P2PLS) using Content Distribution Network (CDN). In contrast, OwnStream is an OpenSource system.

We now compare against an Energy-Efficient Mobile P2P Video Streaming [26]. Their goal is to minimize and balance the energy consumption of participating devices in the video streaming session. Similarly, our goal is to provide the best quality of video streaming across all participants of our video streaming session balancing the traffic in the network using data structures to provide congestion control (maximizing network utilization). Wichtlhuber et al. have energy consumption as its main goal, we can extend our vision and minimize the utilization of mobile devices to avoid degradation of our network and maximize battery life in mobile devices.

We based much of our work on PROMISE and CollectCast [23], [1], [2]. However, there are many important differences. For instance, we assume constant bandwidth for each peer (e.g. we do not have the notion of partial bandwidth contribution, we use all bandwidth available on the WebRTC data channel), and there is only one *broadcaster* for every stream. Available streams are provided by the *supervisor* to the *receivers*. In [1], [2] they exploit the properties of the underlying network in which one receiver can collect data from multiple senders. PROMISE is independent of the underlying P2P network, therefore, PROMISE can be deployed using Pastry [21], Chord [27], and CAN [28]. OwnStream depends on WebSockets and WebRTC architecture, object look-up is done by the *supervisor*. Furthermore, we implemented our own object lookup, called *Stream Manager*.

VII. Conclusion

We presented *OwnStream*, an open source peer-to-peer distributed live media streaming system which aims to provide an alternative to current CDN based streaming systems. We exposed the main idea of creating point to point communication channels between viewers in a tree virtual topology, where failures (disconnections) cause the topology to break, although it can be quickly

restored using a supervisor component that manages all viewers participating in such stream. We also exhibited several experiments to evaluate the streaming quality of the system under five different failure distributions and rates. We concluded that the average stalled time in the overall topology is around one second, mostly milliseconds.

Ownstream offers authentication, privacy, and integrity of media chunks using public key infrastructure, a derivation of SSL protocol (e.g. DTLS) via WebRTC data channels, and a modified version of OTDP (One Time Digest Protocol) respectively. Our final contribution is to have an alternate way to stream media to a large number of clients without incurring in costly and large CDN infrastructure. We also highlighted other potential uses such as media content distribution in mobile apps, and other systems that require large infrastructures, such as software updates and heavily transited web pages.

REFERENCES

[1] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "PROMISE: Peer-to-Peer Media Streaming Using CollectCast," *Proceedings of the eleventh ACM international conference on Multimedia - MULTIMEDIA '03*, p. 45, 2003.

[2] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev, "CollectCast: A peer-to-peer service for media streaming," *Multimedia Systems*, vol. 11, no. 1, pp. 68–81, 2005.

[3] N. Modadugu and E. Rescorla, "The Design and Implementation of Datagram TLS," *Proceedings of ISOC NDSS*, p. 14, 2004.

[4] WebSockets, "WebSockets."

[5] I. Grigorik, *High Performance Browser Networking*. 2013.

[6] P. R. S. Loreto Salvatore, *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*. 2014.

[7] SocketCluster.io, "SocketCluster.io."

[8] "The WebSocket Protocol RFC6455," 2011.

[9] "Session Traversal Utilities for NAT (STUN)," 2008.

[10] "Traversal Using Relays around NAT (TURN)," 2010.

[11] "SDP: Session Description Protocol," 2006.

[12] "Datagram Transport Layer Security," 2006.

[13] "Stream Control Transmission Protocol," 2007.

[14] "The Secure Real-time Transport Protocol (SRTP)," 2004.

[15] PeerJS, "PeerJS,"

[16] NPM, "Node Package Manager."

[17] "Electron."

[18] "Media Source Extensions."

[19] "Chromium."

[20] "WebM," 2014.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Middleware 2001*, vol. 2218, no. November 2001, pp. 329–350, 2001.

[22] R. Ferreira, R. Ferreira, S. Jagannathan, S. Jagannathan, a. Grama, and a. Grama, "Locality in structured peer-to-peer networks," *Journal of Parallel and Distributed Computing*, vol. 66, no. 2, pp. 257–273, 2006.

[23] A. Habib, D. Xu, M. Atallah, and B. Bhargava, "Verifying Data Integrity in Peer-to-Peer Video Streaming," *Cerias.Purdue.Edu*, pp. 1–11, 2005.

[24] T. Sandholm, B. Magnusson, and B. a. Johnsson, "On-Demand WebRTC Tunneling in Restricted Networks," *arXiv.org*, vol. cs.NI, 2013.

[25] R. Roverso, S. El-Ansary, and S. Haridi, "Peer2View: A peer-to-peer HTTP-live streaming platform," *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012*, pp. 65–66, 2012.

[26] M. Wichtlhuber, J. Rückert, D. Stingl, M. Schulz, and D. Hausheer, "Energy-efficient mobile P2P video streaming," *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012*, pp. 63–64, 2012.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pp. 149–160, 2001.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, vol. 31, no. 4, pp. 161–172, 2001.